Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 332 (2017) 21–38

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Formalisation of Nominal α-equivalence with A and AC Function Symbols [1]

Mauricio Ayala-Rincón[†,‡], Washington de Carvalho-Segundo[‡], Maribel Fernández[*] and Daniele Nantes-Sobrinho[†2]

Departamentos de [†] Matemática e [‡] Ciência da Computação
Universidade de Brasília, Brazil

[*] Department of Informatics
King's College London, England, UK

Abstract

A formalisation of soundness of the notion of α-equivalence in nominal abstract syntax modulo associative (A) and associative-commutative (AC) equational theories is described. Initially, the notion of α-equivalence is specified based on a so called "weak" nominal relation as suggested by Urban in his nominal development in Isabelle/HOL. Then, it is formalised in Coq that this equality is indeed an equivalence relation. After that, general α-equivalence with A and AC function symbols is specified and formally proved to be an equivalence relation. As corollaries, the soundness α-equivalence modulo A and modulo AC is obtained. Finally, an algorithm for checking α-equivalence modulo A and AC is proposed. General α-equivalence problems are log-linearly solved while AC and the combination of A and AC α-equivalence problems have the same complexity as standard first-order approaches. This development is a first step towards verification of nominal matching, unification and narrowing algorithms modulo equational theories in general.

Keywords: Nominal logic; Alpha Equivalence, Equivalence modulo A and AC.

1 Introduction

Matching, unification and, more generally, checking the validity of equational problems involving existential and universal quantification is a fundamental issue in automatic deduction. Roughly speaking, two terms s and t are *syntactically equivalent* (resp. *matchable*, *unifiable*) if $s = t$ (resp. if there is a substitution σ such that $s\sigma = t$, or a substitution that applied to s and t simultaneously makes them equal: $s\sigma = t\sigma$). These notions can be extended to equational theories such as *α-equivalence, commutativity, associativity, idempotence*, etc. More generally, we consider *equivalence, matching* and *unification modulo E*, where E is a set of equational axioms. In particular, α-equivalence plays a fundamental role in the *λ-calculus* [5] where it captures the notion of irrelevance of the names used as *bound variables.*

[1] Partially supported by CNPq Universal grant 476952/2013-1.

[2] Email: ayala@unb.br, wtonribeiro@gmail.com, maribel.fernandez@kcl.ac.uk, dnantes@mat.unb.br

http://dx.doi.org/10.1016/j.entcs.2017.04.003

Adequate manipulation of bound variables was a main motivation for the development of Nominal Logic [20] and other formal developments including, *nominal unification* [2,9,19,25,26], that is, unification modulo $\approx_\alpha$, *nominal rewriting* [16,17,18], *deduction systems* [11], *programming languages* [8,22,23] and *reasoning frameworks* [3,24]. In nominal syntax instead of variables one uses *atoms* that are differentiated by their names and used to build abstractions. Additionally, the notion of *freshness* is made explicit through inference rules that define whether atoms are *free* or not in a nominal term. *Renaming* of variables is defined through *swappings* of atoms that are essential components of *permutations acting* over terms. Finally, the notion of α-equivalence is made concrete through inference rules that specify whether, under some freshness constraints, terms are α-equivalent or not. This differs from the usual treatment in frameworks such as the λ-calculus, where α-equivalence is implicitly abstracted through assumptions such as Barendregt's variable convention [5].

The best known and most complete formal development of nominal syntax was specified in Isabelle/HOL by Urban et al. ([25,26]): firstly, a relation $\approx_\alpha$ is specified and proved to be sound, that is, proved to be an equivalence relation; secondly, a nominal unification algorithm is specified, which uses α-equivalence, and verified to be correct and complete. In particular, in [25], Urban describes in detail how to prove that the nominal $\approx_\alpha$ relation is in fact an equivalence relation using an intermediate *weak* α-relation denoted as $\sim_\omega$.

Contribution. A formalisation in Coq of the soundness of α-equivalence in nominal syntax is described. The distinguishing feature of this development is that for the first time, to the best of our knowledge, we advance further and also check nominal α-equivalence with A and AC operators. The development can be enlarged with other equational theories. The main steps of the formalisation are described below.

- Initially, the notions of α-equivalence $\approx_\alpha$ and the *weak equivalence* $\sim_\omega$ are specified following Urban's proof style [25]. Then it is formally proved that $\sim_\omega$ is an equivalence relation. Using $\sim_\omega$ it is then proved that the specified notion of α-equivalence is sound. Although this property is usually taken for granted, its formalisation is not straightforward, since it relies on a non trivial induction on terms in which the induction hypothesis cannot be directly established for *convenient (α) renaming of proper sub-terms of the term to which the induction is applied.* Other crucial, non-trivial properties needed are: preservation of freshness, equivariance of $\approx_\alpha$, preservation of permutation action.
- An α-equivalence relation for terms with A and AC operators, denoted $\approx_{\{A,AC\}}$, is specified and proved sound. The soundness of α-equivalence modulo A ($\approx_{\alpha,A}$) and modulo AC ($\approx_{\alpha,AC}$) is inferred from the soundness of $\approx_{\{A,AC\}}$. These relations are specified in a parameterised manner, which will simplify the treatment and combination of α-equivalence with other equational theories. Function symbols are annotated to indicate whether they are A or AC. The relation $\approx_{\{A,AC\}}$ uses the rules of α-equivalence and it is proved that restricting it to α-equivalence corresponds to $\approx_\alpha$. Thus, using correctness of $\approx_\alpha$, the relation $\approx_{\{A,AC\}}$ is checked by applying the algebraic properties of A and AC operators and, in addition

properties of *preservation of freshness* and *equivariance* for $\approx_{\{A,AC\}}$.

- An algorithm, based on the Coq specifications, for deciding $\approx_{\{A,AC\}}$ is given. When checking equivalence, the decision whether or not to apply nominal inference rules specialised for A or AC symbols is done in a natural manner using the function symbol annotations. Assuming a pre-computation of the flat form of terms headed with function symbols, it is proved that the cost of deciding α-equivalence only modulo A is log-linear on the size of the problem, whereas α-equivalence modulo AC behaves as the algorithm presented by Benanav, Kapur and Narendran [6] for the case of pure AC-equivalence.

Related Work. Equational problems have been extensively explored since the early development of modern abstract algebra (see, e.g., the E-unification survey by Baader et al [4]). Specifically, regarding AC unification and according to Boudet, Contejan and Devie [7] "AC unification is a main issue in term rewriting and automated deduction in general". The treatment given to the problem of deciding AC equality in usual first-order syntax reduces to the problem of searching for a perfect matching in a bipartite graph, as shown in [6].

In addition to the Isabelle/HOL formalisation, there are also formal nominal developments in Coq and PVS. Ayala-Rincón, Fernández and Rocha-Oliveira [2] formalised $\approx_\alpha$-equivalence in PVS without using the auxiliary relation $\sim_\omega$, following the proof sketches proposed in [16], and provided a formalisation of correctness of a nominal unification algorithm. Aydemir, Bohannon and Weirich developed nominal reasoning techniques in Coq [3]. In contrast to the current development, that approach does not take into account a formal verification of nominal equality, because it identifies α-equivalent terms by indexation of the occurrences of bound variables as natural numbers according to their position in the term.

Recently, Copello et al. [12] presented a nominal approach, based essentially on nominal swapping and freshness, used to deal in a concrete manner with α-conversion in the λ-calculus. This was used to formalise in `Agda` principles of α-structural induction and recursion through this nominal concrete implementation of Barendregt's variable convention. Also, Schmidt-Schauss et al. [21] presented nominal unification and matching algorithms for λ-expressions with a recursive let instruction. They proved that both problems are NP-complete and transferring the method, they proved that nominal commutative unification is also NP-complete.

Outline. Section 2 presents necessary background on nominal abstract syntax. Sections 3 and 4 respectively present the formalisations of soundness of α-equivalence and its version with A and AC operators. Before concluding, Section 5 discusses algorithms for deciding $\approx_{\{A,AC\}}$, extracted from the Coq specification, that is available at `https://github.com/wtonribeiro/nominal-ac`.

2 Nominal Syntax

This section introduces nominal syntax following [25,16].

Given a signature Σ of function symbols and countably infinite sets $\mathcal{V}$ and $\mathcal{A}$ of *variables* and *atoms*, respectively, the set $\mathcal{T}(\Sigma, \mathcal{A}, \mathcal{V})$ of *nominal terms* is generated

by the following grammar:

$$s, t \ ::= \ \langle\rangle \ \mid \ \mathtt{a} \ \mid \ [a]t \ \mid \ \langle s, t\rangle \ \mid \ f_k^E\, t \ \mid \ \pi.X$$

Atoms only differ in their names, so for atoms a and b the expression $a \neq b$ is redundant. A *permutation* is a bijection on $\mathcal{A}$ with a finite domain. A *swapping* is defined as a pair of atoms $(a\, b)$ and a *permutation* π is represented by a finite list of *swappings* of the form $(a_1\, b_1) :: \ldots :: (a_n\, b_n) :: \mathit{nil}$, where *nil* denotes the identity permutation. The reverse list of swapings representing π, corresponds to π^{-1}, the inverse of π. The composition of permutations π and π' is denoted as $\pi' \oplus \pi$. Unary permutations $(a\, b) :: \mathit{nil}$ will be abbreviated as $(a\, b)$. A variable $X \in \mathcal{V}$ as a term object should always be decorated by some permutation π *suspended* on X, $\pi.X$. For brevity, terms of the form $\mathit{nil}.X$ will be written as X.

Permutations *act* on nominal terms, but suspend over variables. The *empty tuple* or *unit* is denoted as $\langle\rangle$ and non empty tuples are built using *pairs* of terms of the form $\langle s, t\rangle$, where s and t might be also pairs. Notice that this syntax does not allow construction of unary tuples. The notation `a` represents the atom a as a term object. $[a]t$ is an *abstraction* of an atom a in a term t. The notation $f_k^E\, t$ represents the *application* of $f_k^E \in \Sigma$ to t. The scripts E and k in the function symbol f_k^E are respectively used to distinguish the equational properties of the function symbol and the indexation of the function symbol between the class of operators with the same equational properties. These scripts will be omitted when no confusion arises.

```
Inductive Atom : Set :=
atom : nat → Atom.
Inductive Var : Set :=
var : nat → Var.
Definition Perm :=
list (Atom × Atom).
```

```
Inductive term : Set :=
  | Ut : term
  | At : Atom → term
  | Ab : Atom → term → term
  | Pr : term → term → term
  | Fc : nat → nat → term → term
  | Su : Perm → Var → term
```

In the specification the grammar is written as above. Operators `Ut, At, Ab, Pr, Fc` and `Su` specify the unit, atoms as term objects, abstractions, pairs, function applications and suspended variables, respectively. For the `Fc` constructor, the first and second `nat` arguments represent the super and subscripts of the applied function symbol. In the specification, the function symbols f_j^A and f_k^{AC} are represented respectively by `Fc 0 j` and `Fc 1 k`, both having type `term → term`. All other superscripts are representing the empty equational theory.

An atom as an object term `a`, is written in Coq as `(At a)`. When necessary, this syntax is used in the pseudo-code describing the specification, otherwise standard nominal syntax will be adopted. Notice that although in nominal syntax two atoms a and d are different by definition, `(At a)` and `(At d)` could be the same atom, since in the Coq specification a and d are used as meta-variables ranging over atoms.

Definition 2.1 The **action of a permutation** over terms is specified as the homeomorphic extension of the action of lists of swappings over single atoms:

$$nil \cdot a := a$$

$$((c\ d) :: \pi') \cdot a := \begin{cases} \texttt{if } c = a \texttt{ then } \pi' \cdot d \\ \texttt{else} \begin{cases} \texttt{if } d = a \texttt{ then } \pi' \cdot c \\ \texttt{else } \pi' \cdot a \end{cases} \end{cases}$$

$$\pi \cdot t := \begin{cases} \pi \cdot \langle\rangle & \to \langle\rangle \\ \pi \cdot \texttt{a} & \to (\texttt{At } \pi \cdot a) \\ \pi \cdot f_k^E\, t & \to f_k^E\, (\pi \cdot t) \\ \pi \cdot \langle u, v\rangle & \to \langle \pi \cdot u, \pi \cdot v\rangle \\ \pi \cdot ([a]t) & \to [\pi \cdot a](\pi \cdot t) \\ \pi \cdot (\pi' \,.\, X) & \to (\pi' \oplus \pi) \,.\, X \end{cases}$$

The *action of a permutation* over an atomic term object $\texttt{a}$, e.g., $nil \cdot \texttt{a}$, gives as result a term. This is specified as $nil \cdot (\texttt{At}\ a)$, which gives as result $\texttt{a}$, that is, $(\texttt{At}\ a)$, and not the atom a. The action of the permutation π over the suspended variable $\pi'.X$ gives as result the term $\pi \cdot (\pi'.X) = (\pi' \oplus \pi).X$. Notice that permutation composition works in the opposite direction.

The permutation $(a\ b) :: \pi$ acting over the term $[a]\langle \texttt{b}, \pi'.X\rangle$ will have as result $[\pi \cdot b]\langle \texttt{At}\ (\pi \cdot a), (\pi' \oplus ((a\ b) :: \pi)).X\rangle$.

The native notion of equality on nominal terms is α-equivalence, which is defined using swappings and a notion of freshness. A *freshness constraint* is a pair $a \# t$ of an atom and a nominal term t. Intuitively, $a \# t$ means that a is fresh in t, that is, if a occurs in t then it must do so under an abstractor $[a]$. An *α-equality constraint* is a pair $s \approx_\alpha t$ of two terms s and t. A *freshness context*, is a set of *freshness constraints*. ∇ will range over freshness contexts. A *freshness judgement* is a tuple of the form $\nabla \vdash a \# t$ whereas an *α-equivalence judgement* is a tuple of the form $\nabla \vdash s \approx_\alpha t$.

Table 1
Rules for the freshness relation

$$\frac{}{\nabla \vdash a \# \langle\rangle}\ \texttt{[\#-ut]} \qquad \frac{}{\nabla \vdash a \# \texttt{b}}\ \texttt{[\#-at]} \qquad \frac{\nabla \vdash a \# t}{\nabla \vdash a \# f_k^E\, t}\ \texttt{[\#-fc]}$$

$$\frac{\nabla \vdash a \# t_1 \qquad \nabla \vdash a \# t_2}{\nabla \vdash a \# \langle t_1, t_2\rangle}\ \texttt{[\#-pr]} \qquad \frac{}{\nabla \vdash a \# [a]t}\ \texttt{[\#-ab}_1\texttt{]}$$

$$\frac{\nabla \vdash a \# t}{\nabla \vdash a \# [b]t}\ \texttt{[\#-ab}_2\texttt{]} \qquad \frac{\pi^{-1} \cdot a \# X \in \nabla}{\nabla \vdash a \# \pi.X}\ \texttt{[\#-su]}$$

The *derivable* freshness and α-equivalence judgements are defined by the rules in Tables 1 and 2. We write $ds(\pi, \pi')\#X$ as an abbreviation of $\{a \# X \mid a \in ds(\pi, \pi')\}$, where $ds(\pi, \pi') = \{a \mid \pi \cdot a \neq \pi' \cdot a\}$ is the set of atoms where π and π' differ (the *difference set*). A set $\mathcal{P}$ of constraints is called a *problem*. We write $\nabla \vdash \mathcal{P}$ when proofs of the judgment $\nabla \vdash P$ exist for each $P \in \mathcal{P}$, using rules of Tables 1 and 2.

The rules for abstractions and suspensions are the interesting ones. For example, $\nabla \vdash a \# \langle [a](\langle \texttt{a}, \texttt{b}\rangle), \pi.X\rangle$ can be derived only if $\pi^{-1} \cdot a \# X$ is in ∇. There are two rules for abstractions in Table 2: $\texttt{[}\approx_\alpha\texttt{-ab}_1\texttt{]}$ and $\texttt{[}\approx_\alpha\texttt{-ab}_2\texttt{]}$. The latter, for abstractions built with different atoms, swaps the atoms in one of the abstractions, provided the atom is fresh.

Table 2
Rules for α-equivalence

$$\frac{}{\nabla \vdash \langle\rangle \approx_\alpha \langle\rangle}\ [\approx_\alpha\texttt{-ut}] \qquad \frac{}{\nabla \vdash \mathtt{a} \approx_\alpha \mathtt{a}}\ [\approx_\alpha\texttt{-at}] \qquad \frac{\nabla \vdash t \approx_\alpha t'}{\nabla \vdash f_k^E\, t \approx_\alpha f_k^E\, t'}\ [\approx_\alpha\texttt{-fc}]$$

$$\frac{\nabla \vdash t_1 \approx_\alpha t_1' \qquad \nabla \vdash t_2 \approx_\alpha t_2'}{\nabla \vdash \langle t_1, t_2\rangle \approx_\alpha \langle t_1', t_2'\rangle}\ [\approx_\alpha\texttt{-pr}] \qquad \frac{\nabla \vdash t \approx_\alpha t'}{\nabla \vdash [a]t \approx_\alpha [a]t'}\ [\approx_\alpha\texttt{-ab}_1]$$

$$\frac{\nabla \vdash t \approx_\alpha (a\ b)\cdot t' \quad \nabla \vdash a \# t'}{\nabla \vdash [a]t \approx_\alpha [b]t'}\ [\approx_\alpha\texttt{-ab}_2] \qquad \frac{ds(\pi,\pi') \# X \subseteq \nabla}{\nabla \vdash \pi.X \approx_\alpha \pi'.X}\ [\approx_\alpha\texttt{-su}]$$

3 Formalisation of soundness of the $\approx_\alpha$ relation

Using the Coq specification, `alpha_equiv` (that is, $\approx_\alpha$ of Table 2) was formally proved to be an equivalence relation. This section describes the formalisation.

Standard proofs use a measure on terms to prove that $\approx_\alpha$ is symmetric and then prove that $\approx_\alpha$ is transitive [2,16,19,26]. We use an alternative proof proposed by Urban in [25]: initially a so called " weak" equivalence relation $\sim_\omega$ is defined, as given in Table 3. Afterwards, $\sim_\omega$ is proved to be an equivalence relation, which is straightforward and gives an intermediate transitivity result for $\approx_\alpha$: $\nabla \vdash t_1 \approx_\alpha t_2$ and $t_2 \sim_\omega t_3$ implies $\nabla \vdash t_1 \approx_\alpha t_3$. Finally, this result is used in conjunction with some auxiliary lemmas to prove firstly the transitivity and then the symmetry of $\approx_\alpha$. The final part of the formalisation relies on three main auxiliary lemmas:

- *Freshness preservation of* $\approx_\alpha$: $\nabla \vdash a \# t$ and $\nabla \vdash t \approx_\alpha t'$ imply $\nabla \vdash a \# t'$;
- *Equivariance of* $\approx_\alpha$: $\nabla \vdash t \approx_\alpha t'$ implies $\nabla \vdash \pi \cdot t \approx_\alpha \pi \cdot t'$;
- *Invariance of* $\approx_\alpha$ *under the action of permutations*: $(\forall a \in ds(\pi,\pi'), \nabla \vdash a \# t)$ iff $\nabla \vdash \pi \cdot t \approx_\alpha \pi' \cdot t$.

Table 3
Rules for weak α-equivalence

$$\frac{}{\langle\rangle \sim_\omega \langle\rangle}\ [\sim_\omega\texttt{-ut}] \qquad \frac{}{a \sim_\omega a}\ [\sim_\omega\texttt{-at}] \qquad \frac{t \sim_\omega t'}{f_k^E t \sim_\omega f_k^E t'}\ [\sim_\omega\texttt{-fc}]$$

$$\frac{t_1 \sim_\omega t_1' \qquad t_2 \sim_\omega t_2'}{\langle t_1, t_2\rangle \sim_\omega \langle t_1', t_2'\rangle}\ [\sim_\omega\texttt{-pr}] \qquad \frac{t \sim_\omega t'}{[a]t \sim_\omega [a]t'}\ [\sim_\omega\texttt{-ab}] \qquad \frac{ds(\pi,\pi') = \emptyset}{\pi \cdot X \sim_\omega \pi' \cdot X'}\ [\sim_\omega\texttt{-su}]$$

For checking α-equivalence modulo A and AC, $\approx_{\{A,AC\}}$, one uses soundness of $\approx_\alpha$. Thus, one could adopt any approach for checking $\approx_\alpha$ maintaining the approach for checking $\approx_{\{A,AC\}}$. More specifically, we specify an inductive relation `equiv`(S), where S is a set of indices, each one associated with a different equational theory. In particular, the relation `equiv`$(\emptyset)$ excludes from the specification of `equiv`, all specialised inference rules for any equational theory. The relation `equiv`$(\emptyset)$ is formally proved to be equivalent to the relation $\approx_\alpha$: $\nabla \vdash t \approx_\alpha t' \Leftrightarrow$ `equiv`$(\emptyset)(\nabla, t, t')$.

Lemma 3.1 (Intermediate transitivity for $\approx_\alpha$ with $\sim_\omega$) *If* $\nabla \vdash t_1 \approx_\alpha t_2$ *and* $t_2 \sim_\omega t_3$ *then* $\nabla \vdash t_1 \approx_\alpha t_3$.

The proof is by induction on $\approx_\alpha$. It uses basic properties of nominal terms and expansions of the inference rules in the definition of $\approx_\alpha$, except for the rule $[\approx_\alpha \text{-ab}_2]$ whose analysis requires the application of the equivariance property for $\sim_\omega$ and preservation of freshness under $\sim_\omega$. Namely, in the inductive step of this case one has as premises $\nabla \vdash t_1 \approx_\alpha (a\,b)\,t_2$, $\nabla \vdash [a]t_1 \approx_\alpha [b]t_2$, $[b]t_2 \sim_\omega t_3$, $\nabla \vdash a \# t_2$ and as IH: for all t_0, $(a\,b)\,t_2 \sim_\omega t_0$ implies $\nabla \vdash t_1 \approx_\alpha t_0$; and one needs to conclude that $\nabla \vdash [a]t_1 \approx_\alpha t_3$ (some non relevant premises are omitted). By properties of $\sim_\omega$, $[b]t_2 \sim_\omega t_3$, t_3 should be of the form $[b]t_3'$. Thus, one needs to conclude that $\nabla \vdash [a]t_1 \approx_\alpha [b]t_3'$. Additionally, one has $t_2 \sim_\omega t_3'$ and, by equivariance of $\sim_\omega$, it follows that $(a\,b)t_2 \sim_\omega (a\,b)\,t_3'$. Also, by preservation of freshness under $\sim_\omega$ one has $\nabla \vdash a \# t_3'$. Instantiating the IH with $(a\,b)\,t_3'$, one has that $\nabla \vdash t_1 \approx_\alpha (a\,b)\,t_3'$. From this, applying rule $[\approx_\alpha \text{-ab}_2]$, one finally concludes that $\nabla \vdash [a]t_1 \approx_\alpha [b]t_3'$.

Lemma 3.2 (Freshness preservation of $\approx_\alpha$) *If* $\nabla \vdash a \# t$ *and* $\nabla \vdash t \approx_\alpha t'$ *then* $\nabla \vdash a \# t'$.

The proof is by induction on $\approx_\alpha$. The interesting case is the analysis of the $[\approx_\alpha\text{-}\mathtt{ab}_2]$ rule, whose hypotheses are $a_0 \neq b_0$, $\nabla \vdash t \approx_\alpha (a_0\,b_0)\,t'$, $\nabla \vdash a_0 \# t'$, and $\nabla \vdash a \# [a_0]t$. By IH $\nabla \vdash a \# t \Rightarrow \nabla \vdash a \# (a_0\,b_0)\,t'$. One should prove that $\nabla \vdash a \# [b_0]t'$. For doing this, the three cases: $a = a_0$, $b_0 = a \neq a_0$ and $b_0 \neq a \neq a_0$ should be analysed. The difficult case is the last one, which is solved by application of the $[\#\mathtt{ab}_2]$ rule with the use of a technical lemma about the freshness relation.

Lemma 3.3 (Equivariance of $\approx_\alpha$) *If* $\nabla \vdash t \approx_\alpha t'$ *then* $\nabla \vdash \pi \cdot t \approx_\alpha \pi \cdot t'$.

The formalisation is by induction on $\approx_\alpha$. The tricky case is when one has as hypotheses $a \neq b$, $\nabla \vdash t \approx_\alpha (a\,b)\,t'$ and $\nabla \vdash a \# t'$. The IH is $\nabla \vdash \pi \cdot t \approx_\alpha \pi \cdot ((a\,b)\,t')$. It should be proved that $\nabla \vdash \pi \cdot ([a]t) \approx_\alpha \pi \cdot ([b]t')$. Applying the definition of the permutation action and the $[\approx_\alpha\text{-}\mathtt{ab}_2]$ rule, three subgoals have to be proved: $\pi \cdot a \neq \pi \cdot b$, $\nabla \vdash (\pi \cdot t) \approx_\alpha ((\pi \cdot a)\,(\pi \cdot b))\,(\pi \cdot t')$ and $\nabla \vdash (\pi \cdot a) \# (\pi \cdot t')$. The first and the last are trivially solved by technical lemmas. For the second sub goal, Lem. 3.1 instantiating t_2 with $\pi \cdot ((a\,b)\,t')$ is applied. Then one of the new subgoals is the IH and the other one is $(\pi \cdot ((a\,b)\,t)) \sim_\omega ((\pi \cdot a)\,(\pi \cdot b))\,(\pi \cdot t')$. The latter is an instance of a technical lemma about the distribution of a permutation among swappings.

The next result will be applied in the proof of Lem. 3.5 and in proofs related with symmetry, as preservation of α-equivalence under swappings, e.g., $\nabla \vdash (a\,b) \cdot t \approx_\alpha (c\,d) \cdot t$ with $\nabla \vdash a, b, c, d \# t$.

Lemma 3.4 (Invariance of terms under $\approx_\alpha$ and action of permutations) $(\forall a \in ds(\pi, \pi'), \nabla \vdash a \# t)$ *iff* $\nabla \vdash \pi \cdot t \approx_\alpha \pi' \cdot t$.

This lemma is proved by induction on t under arbitrary permutations.

Lemma 3.5 (Second intermediate transitivity lemma) *If* $\nabla \vdash t_1 \approx_\alpha t_2$ *and* $\nabla \vdash t_2 \approx_\alpha \pi \cdot t_2$ *then* $\nabla \vdash t_1 \approx_\alpha \pi \cdot t_2$.

The proof is by induction on $\approx_\alpha$ in the hypothesis $\nabla \vdash t_1 \approx_\alpha t_2$. The interesting case is for abstractions, that is $t_1 = [a]t_1'$ and $t_2 = [b]t_2'$. Several cases are to be considered according to whether a and b, a and $\pi \cdot a$, b and $\pi \cdot b$ as well as a and $\pi \cdot b$ are or not equal.

Lemma 3.6 (Reflexivity of $\approx_\alpha$) $\nabla \vdash t \approx_\alpha t$.

This lemma is proved by routine induction over the structure of t.

Lemma 3.7 (Transitivity of $\approx_\alpha$) *If* $\nabla \vdash t_1 \approx_\alpha t_2$ *and* $\nabla \vdash t_2 \approx_\alpha t_3$ *then* $\nabla \vdash t_1 \approx_\alpha t_3$.

The formalisation is by induction in $\nabla \vdash t_1 \approx_\alpha t_2$ with generalisation of t_3. The difficult case occurs when $t_1 = [a]t_1'$, $t_2 = [b]t_2'$ and $t_3 = [c]t_3'$, with $a \neq b \neq c \neq a$. The IH is given as $\forall t_0, \nabla \vdash t_2' \approx_\alpha t_0 \Rightarrow \nabla \vdash t_1' \approx_\alpha t_0$, and the other hypotheses are: $\nabla \vdash t_1' \approx_\alpha (a\,b)\,t_2'$, $\nabla \vdash a \# t_2'$, $\nabla \vdash t_2' \approx_\alpha (b\,c)\,t_3'$ and $\nabla \vdash b \# t_3'$. It should be concluded that $\nabla \vdash [a]t_1' \approx_\alpha [c]t_3'$.

Applying the rule $[\approx_\alpha\texttt{-ab}_2]$ to the goal one obtains the subgoals $\nabla \vdash a \# t_3'$ and $\nabla \vdash t_1' \approx_\alpha (a\,c)\,t_3'$. The former is proved by Lem. 3.2. Applying IH over the latter subgoal, it remains to prove $\nabla \vdash (a\,b)\,t_2' \approx_\alpha (a\,c)\,t_3'$. So, it is needed to prove the intermediate statement $\nabla \vdash [(b\,c) :: (a\,b)] \cdot t_3' \approx_\alpha [(b\,c) :: (a\,b) :: (b\,c)] \cdot t_3'$, that is possible by application of Lem. 3.4. Manipulating swappings and using Lem. 3.5 one infers $\nabla \vdash (a\,b)\,t_2' \approx_\alpha [(b\,c) :: (a\,b) :: (b\,c)] \cdot t_3'$. Finally, applying Lem. 3.1 with $t_2 := [(b\,c) :: (a\,b) :: (b\,c)] \cdot t_3'$ only remains to prove that $[(b\,c) :: (a\,b) :: (b\,c)] \cdot t_3' \sim_\omega (a\,c)\,t_3'$, that can be done using properties of $\sim_\omega$ such as its equivalence and equivariance.

Lemma 3.8 (Symmetry of $\approx_\alpha$) *If* $\nabla \vdash t \approx_\alpha t'$ *then* $\nabla \vdash t' \approx_\alpha t$.

The proof is by induction on $\approx_\alpha$ over $\nabla \vdash t \approx_\alpha t'$. The non-trivial case is when $a \neq b$ and the hypotheses are $\nabla \vdash t_0 \approx_\alpha (a\,b)\,t_0'$, $\nabla \vdash a \# t_0'$ and $\nabla \vdash (a\,b)\,t_0' \approx_\alpha t_0$, with the subgoal $\nabla \vdash [b]t_0' \approx_\alpha [a]t_0$. This is proved by application of the rule $[\approx_\alpha\texttt{-ab}_2]$ and then by a double application of Lem. 3.7 instantiated with $t_2 := (a\,b)\,t_0$ and $t_2 := [(a\,b) :: (a\,b)] \cdot t_0$. The remaining subgoals are treated using Lem. 3.3.

4 Formalising soundness of $\approx_{\{A,AC\}}$, $\approx_{\alpha,A}$ and $\approx_{\alpha,AC}$

The generic relation $\texttt{equiv}(S)$ mentioned at the end of Sec. 3 takes into account A and AC function symbols if $0 \in S$ and $1 \in S$, respectively.

Namely, $\texttt{equiv}(\{0\})$, $\texttt{equiv}(\{1\})$ and $\texttt{equiv}(\{0,1\})$ choose the specialised inductive rules in the definition of `equiv` for the relation $\approx_\alpha$ modulo A, AC and A combined with AC function symbols, respectively. In this way one builds the relations $\approx_{\alpha,A}$, $\approx_{\alpha,AC}$ and $\approx_{\{A,AC\}}$. Function symbols with superscripts 0 and 1 will be interpreted as A and AC operators respectively, only when the parameter S includes 0 or/and 1. Using the parameter $S = \{0\}$ and function symbols with arbitrary superscripts, including 0, and only function symbols with superscript 0, respectively *general* and *elementary* α-equational modulo A problems are expressed ([4]). The same happens for AC problems. The formalisations given here are for

general α-equivalence problems. For simplicity, instead 0 and 1 we will use A and AC in the sequel.

4.1 Operations over tuples

The inductive rules for A and AC operators in the definition of the relation $\approx_{\{A,AC\}}$ use three auxiliary operators that deal with arguments of function symbols. Arguments of a function symbol f are terms or tuples built using the constructor for pairs and the arguments of terms headed by the same function symbol f. These operators, specified as in Fig. 1, extract the relevant information of the arguments to which a(n A or AC) symbol f is applied and specify the *length or number of arguments*, $\|t\|_f$, and the *selection* and *deletion of the* i^{th} *argument*, respectively, $t_{(i)_f}$ and $t_{[\star i]_f}$.

$$\|t\|_f := \begin{cases} \langle s,u\rangle \to \|s\|_f + \|u\|_f \\ g\ s \ \to \begin{cases} \texttt{if}\ g = f \\ \texttt{then}\ \|s\|_f \\ \texttt{else}\ 1 \end{cases} \\ \_ \ \to 1 \end{cases} \qquad t_{(i)_f} := \begin{cases} \langle s,u\rangle \to \begin{cases} \texttt{if}\ i \leq \|s\|_f\ \texttt{then}\ s_{(i)_f} \\ \texttt{else}\ u_{(i-\|s\|_f)_f} \end{cases} \\ g\ s \ \to \begin{cases} \texttt{if}\ g = f\ \texttt{then}\ s_{(i)_f} \\ \texttt{else}\ g\ s \end{cases} \\ \_ \ \to t \end{cases}$$

$$t_{[\star i]_f} := \begin{cases} \langle s,u\rangle \to \begin{cases} \texttt{if}\ i \leq \|s\|_f\ \texttt{then} \begin{cases} \texttt{if}\ \|s\|_f = 1\ \texttt{then}\ u \\ \texttt{else}\ \langle s_{[\star i]_f}, u\rangle \end{cases} \\ \texttt{else} \begin{cases} \texttt{if}\ \|u\|_f = 1\ \texttt{then}\ s \\ \texttt{else}\ \langle s, u_{[\star(i-\|s\|_f)]_f}\rangle \end{cases} \end{cases} \\ g\ s \ \to \begin{cases} \texttt{if}\ \|g\ s\|_f = 1\ \texttt{then}\ \langle\rangle \\ \texttt{else}\ g\,(s_{[\star i]_f}) \end{cases} \\ \_ \ \to \langle\rangle \end{cases}$$

Fig. 1. Specification of operators for the length of the tuple or arguments, selection and deletion of the i^{th} argument regarding the function symbol f

To simplify notation, the scripts of f will be omitted in these operators when clear from the context. The behaviour of these operators is illustrated below

Example 4.1 For the number of arguments. • $\|f\langle\rangle\|_f = \|\langle\rangle\|_f = 1$. • $\|f\,\langle \mathtt{a},\mathtt{b}\rangle\|_f = \|\langle \mathtt{a},\mathtt{b}\rangle\|_f = 2$, but $\|g\,\langle \mathtt{a},\mathtt{b}\rangle\|_f = 1$; • $\|f\,\langle [a](\pi\cdot X), f\,\langle \mathtt{b}, g\,\langle \mathtt{a}, f\,\langle \mathtt{a},\mathtt{b}\rangle\rangle\rangle\rangle\|_f = \|[a](\pi\cdot X)\|_f + \|\mathtt{b}\|_f + \|g\,\langle \mathtt{a}, f\,\langle \mathtt{a},\mathtt{b}\rangle\rangle\|_f = 3$.

Example 4.2 For the selection of the i^{th} argument. • $t_{(0)_f} = t_{(1)_f}$ and, if $i > \|t\|_f$ then $t_{(i)_f} = t_{(\|t\|_f)_f}$. • If $\|t\|_f = 1$ and t is not headed by f then $t_{(1)_f} = t$, but $(f\,f\,t)_{(1)_f} = t$; • $(f\,\langle [a](\pi\cdot X), f\,\langle \mathtt{b}, g\,\langle \mathtt{a}, f\,\langle \mathtt{a},\mathtt{b}\rangle\rangle\rangle\rangle)_{(3)_f} = (f\,\langle \mathtt{b}, g\,\langle \mathtt{a}, f\,\langle \mathtt{a},\mathtt{b}\rangle\rangle\rangle)_{(2)_f} = (g\,\langle \mathtt{a}, f\,\langle \mathtt{a},\mathtt{b}\rangle\rangle)_{(1)_f} = g\,\langle \mathtt{a}, f\,\langle \mathtt{a},\mathtt{b}\rangle\rangle$.

Example 4.3 For the deletion of the i^{th} argument. • $t_{[\star 0]_f} = t_{[\star 1]_f}$ and if $i > \|t\|_f$ then $t_{[\star i]_f} = t_{[\star \|t\|_f]_f}$. • If $\|t\|_f = 1$ then $t_{[\star 1]_f} = \langle\rangle$; • $(f\,\langle [a](\pi \cdot X), f\,\langle \mathtt{b}, g\,\langle \mathtt{a}, f\,\langle \mathtt{a}, \mathtt{b}\rangle\rangle\rangle\rangle)_{[\star 2]_f} = f\,\langle [a](\pi \cdot X), (f\,\langle \mathtt{b}, g\,\langle \mathtt{a}, f\,\langle \mathtt{a}, \mathtt{b}\rangle\rangle\rangle)_{[\star 1]_f}\rangle$
$= f\,\langle [a](\pi \cdot X), f\,(g\,\langle \mathtt{a}, f\,\langle \mathtt{a}, \mathtt{b}\rangle\rangle)\rangle$.

It should be clear to the reader that the specification follows the lines of nominal syntax in which function symbols have no fixed arity. Thus for any A or AC symbol it should be interpreted apart what means its application to the unit ($\langle\rangle$) and to a single argument, for instance, with the usual interpretation for operator symbols, $\wedge\langle\rangle$, $\vee\langle\rangle$, $+\langle\rangle$ and $\times\langle\rangle$ might be specified as "false", "true", 0 and 1, respectively.

Using these operators has two advantages: first, no additional data structure (e.g. list, sequence, array) or *flattening* operator is needed to express associativity; second, the approach is generic: the grammar and the given rules can be extended to manipulate operators from various equational theories, in a natural way. If function symbols with different equational properties occur in a term, the specialised inference rules that deal with their equational properties are used. This simplifies the treatment of α-equivalence modulo A and AC, and other equational theories.

In Table 4 a few formalised results are listed, from a much longer list of formalised lemmas related with these operators. These results will be referenced in the description of the lemmas related with E-equivalence and for brevity they are presented free of universal quantifiers.

Table 4
Basic properties of the operators over terms: $\|\_\|_f$, $\_(\_)_f$ and $\_[\star\_]_f$

$\|t\| \geq 1,\ t_{(0)} = t_{(1)},\ t_{[\star 0]} = t_{[\star 1]}$	$i \geq \|t\| \Rightarrow t_{(i)} = t_{(\|t\|)},\ t_{[\star i]} = t_{[\star \|t\|]}$
$\|t\| = 1 \Rightarrow t_{[\star i]} = \langle\rangle$	$\|t\| \neq 1 \Rightarrow \|t_{[\star i]}\| = \|t\| - 1$
$0 < i < j$ or $0 < i < \|t\| \Rightarrow (t_{[\star j]})_{(i)} = t_{(i)}$	$0 < i < j \leq \|t\| \Rightarrow (t_{[\star j]})_{[\star i]} = (t_{[\star i]})_{[\star (j-1)]}$
$0 < i < \|t\|,\ i \geq j \Rightarrow (t_{[\star j]})_{(i)} = t_{(i+1)},\ (t_{[\star j]})_{[\star i]} = (t_{[\star (i+1)]})_{[\star j]}$	

4.2 Extension of $\approx_\alpha$-rules

New rules [`equiv_A`] and [`equiv_AC`] for associativity and commutativity are introduced. These rules will be combined with those from Table 2 for $\approx_\alpha$, with the following modification: [$\approx_\alpha$-`fc`] will be replaced by [`equiv_Fc`] and applies whenever the function symbol f_k^E is such that $E \notin S$; otherwise, when $E = \text{A}$ or $E = \text{AC}$ and $E \in S$, rules [`equiv_A`] or [`equiv_AC`] apply. Thus in the case f is neither an A nor an AC function symbol or $A, AC \notin S$, the behaviour of [`equiv_Fc`] and [$\approx_\alpha$-`fc`] would be exactly the same. These rules define an extended calculus for *general* α-equivalence modulo A and AC ([4]), denoted by the relation $\approx_{\{A,AC\}}$ (specified as `equiv({0,1})`). Other equational theories might be included similarly. Below, $s \approx_{\{A,AC\}} t$ denotes that s and t are α-equivalent modulo A and AC.

Rule [`equiv_A`] applies when the terms compared are headed by the same A function symbol and $A \in S$. It verifies recursively if the first arguments on the left (*lhs*) and right-hand sides (*rhs*) are related by $\approx_{\{A,AC\}}$ as well as the result of applying the root function symbol to the respective tuples without the first argument.

$$\frac{\nabla \vdash t \approx_{\{A,AC\}} t', \qquad E \notin S}{\nabla \vdash f_k^E t \approx_{\{A,AC\}} f_k^E t'} \; [\texttt{equiv\_Fc}]$$

Fig. 2. [equiv_Fc]-rule for $\approx_{\{A,AC\}}$

$$\frac{A \in S, \qquad \begin{array}{l}\nabla \vdash (f_k^A\, s)_{(\mathbf{1})_{f_k^A}} \approx_{\{A,AC\}} (f_k^A\, t)_{(\mathbf{1})_{f_k^A}}, \\ \nabla \vdash (f_k^A\, s)_{[\star\mathbf{1}]_{f_k^A}} \approx_{\{A,AC\}} (f_k^A\, t)_{[\star\mathbf{1}]_{f_k^A}}\end{array}}{\nabla \vdash f_k^A\, s \approx_{\{A,AC\}} f_k^A\, t} \; [\texttt{equiv\_A}]$$

Fig. 3. [equiv_A]-rule for A function symbols

$$\frac{AC \in S, \qquad \begin{array}{l}\nabla \vdash (f_k^{AC}\, s)_{(\mathbf{1})_{f_k^{AC}}} \approx_{\{A,AC\}} (f_k^{AC}\, t)_{(\mathbf{i})_{f_k^{AC}}}, \\ \nabla \vdash (f_k^{AC}\, s)_{[\star\mathbf{1}]_{f_k^{AC}}} \approx_{\{A,AC\}} (f_k^{AC}\, t)_{[\star\mathbf{i}]_{f_k^{AC}}}\end{array}}{\nabla \vdash f_k^{AC}\, s \approx_{\{A,AC\}} f_k^{AC}\, t} \; [\texttt{equiv\_AC}]$$

Fig. 4. [equiv_AC]-rule for AC function symbols

Rule [equiv_AC] behaves similarly to rule [equiv_A]: the fundamental difference is that the first argument on the *lhs* can be compared modulo $\approx_{\{A,AC\}}$ with any arbitrary argument on the *rhs*. If there exists such argument, say the i^{th}, it remains to check that the terms obtained applying the function symbol to the tuples deleting the first and the i^{th} arguments to the right and to the left are related by $\approx_{\{A,AC\}}$.

Example 4.4 $\nabla \;\vdash\; f\,\langle t_1, g^{AC}\,\langle t_2, g^{AC}\langle t_3, t_4\rangle\rangle\rangle \;\approx_{\{A,AC\}}\; f\,\langle t_1, g^{AC}\,\langle\langle t_4, t_3\rangle, t_2\rangle\rangle$, where g is AC, f is a function symbol that allows only α-equivalence and $AC \in S$.

4.3 Checking $\approx_{\{A,AC\}}$, $\approx_{\alpha,A}$ and $\approx_{\alpha,AC}$

An interesting aspect of checking $\approx_{\alpha,AC}$ is that it follows the general lines of formalisation of α-equivalence but using as "weak" relation $\approx_\alpha$ instead of $\sim_\omega$: after proving an intermediate transitivity lemma for $\approx_{\{A,AC\}}$ (Lem. 4.5), one proves *freshness preservation* and *equivariance* (Lemmas 4.6, 4.7) of $\approx_{\{A,AC\}}$ and then, transitivity before symmetry (Lemmas 4.10 4.11). By using the parameter set S on the $\texttt{equiv}(S)$ relation and renaming superscripts of function symbols, one obtains as corollary of the soundness $\approx_{\{A,AC\}}$ the soundness of $\approx_{\alpha,A}$ and $\approx_{\alpha,AC}$.

In addition to preservation of freshness and equivariance, the intermediate transitivity lemma (Lem. 4.5) is relevant to guarantee some key properties on swappings and permutations acting over $\approx_{\{A,AC\}}$-related terms as for instance, $\nabla \vdash t \approx_{\{A,AC\}} (a\,a')\,t' \Rightarrow \nabla \vdash (a'\,a)\,t \approx_{\{A,AC\}} t'$.

Lemma 4.5 (Intermediate transitivity for $\approx_{\{A,AC\}}$ with $\approx_\alpha$) *If* $\nabla \vdash s \approx_{\{A,AC\}} t$ *and* $\nabla \vdash t \approx_\alpha u$ *then* $\nabla \vdash s \approx_{\{A,AC\}} u$.

The formalisation is obtained as follows: after generalisation of u, induction

is applied on deduction rules of $\approx_{\{A,AC\}}$ for $\nabla \vdash s \approx_{\{A,AC\}} t$. In some cases it is required inversion of $\nabla \vdash t \approx_{\{A,AC\}} u$; for instance, in the case in which one has $t = \langle t_1, t_2\rangle$, inversion is applied to obtain that $u = \langle u_1, u_2\rangle$ with $\nabla \vdash t_1 \approx_{\{A,AC\}} u_1$ and $\nabla \vdash t_2 \approx_{\{A,AC\}} u_2$, according to the inference rule $[\approx_\alpha\texttt{-pr}]$.

Lemma 4.6 (Freshness preservation under $\approx_{\{A,AC\}}$) *If $\nabla \vdash a \# s$ and $\nabla \vdash s \approx_{\{A,AC\}} t$ then $\nabla \vdash a \# t$.*

The proof is by induction on $\approx_{\{A,AC\}}$, using some technical results about the freshness relation for dealing with cases related with rules $[\approx_\alpha\texttt{-ab}_1]$ and $[\approx_\alpha\texttt{-ab}_2]$ for the case in which s and t are abstractions.

Lemma 4.7 (Equivariance of $\approx_{\{A,AC\}}$) *If $\nabla \vdash s \approx_{\{A,AC\}} t$ then $\nabla \vdash \pi\cdot s \approx_{\{A,AC\}} \pi\cdot t$.*

Equivariance follows by induction in the inference rules of $\approx_{\{A,AC\}}$. For the case of abstractions, specifically for the case of the rule $[\approx_\alpha\text{-ab}_2]$, Lem. 4.5 is required; indeed, when one has $\nabla \vdash [a]s' \approx_{\{A,AC\}} [b]t'$, initially it is necessary to prove that $\nabla \vdash \pi \cdot s' \approx_{\{A,AC\}} \pi \cdot ((a\ b) \cdot t')$ and $\nabla \vdash \pi \cdot ((a\ b) \cdot t') \approx_\alpha (\pi \cdot a\ \ \pi \cdot b) \cdot (\pi \cdot t')$ and then apply that lemma to obtain $\nabla \vdash \pi \cdot s' \approx_{\{A,AC\}} (\pi \cdot a\ \ \pi \cdot b) \cdot (\pi \cdot t')$.

Lemma 4.8 (Reflexivity of $\approx_{\{A,AC\}}$) $\nabla \vdash t \approx_{\{A,AC\}} t$.

Reflexivity is easily proved by induction on t. The next lemma generalises the way in which arguments used in the rule [equiv_AC] are combined.

Lemma 4.9 (Combination of AC arguments) *If $\nabla \vdash t \approx_{\{A,AC\}} t'$ then $\forall_{(0<i\leq\|t\|_f)}\exists_{(0<j\leq\|t\|_f)}\nabla \vdash t_{(i)_f} \approx_{\{A,AC\}} t'_{(j)_f}$ and $\nabla \vdash t_{[\star i]_f} \approx_{\{A,AC\}} t'_{[\star j]_f}$.*

The proof is by induction on $\|t\|_f$ using simple auxiliary lemmas and properties of the operators $\|t\|_f, t_{(i)_f}$ and $t_{[\star i]_f}$. We explain the particular case for $i = 1$: $\nabla \vdash t \approx_{\{A,AC\}} t' \Rightarrow \exists_{(0<j\leq\|t'\|_f)}, \nabla \vdash t_{(1)_f} \approx_{\{A,AC\}} t'_{(j)_f} \ \wedge\ \nabla \vdash t_{[\star 1]_f} \approx_{\{A,AC\}} t'_{[\star j]}$. The complicated case happens when $\|t\|_f > 2$: after applying the auxiliary lemma for terms $f\,t$ and $f\,t'$ one obtains $\nabla \vdash t_{(1)_f} \approx_{\{A,AC\}} t'_{(i_0)_f}$ and $\nabla \vdash f\,t_{[\star 1]_f} \approx_{\{A,AC\}} f\,t'_{[\star i_0]_f}$, for some i_0. If $i = 1$, the result follows trivially. For $i > 1$, induction applies for the terms $t_0 = f\,t_{[\star 1]_f}$ and $t'_0 = f\,t'_{[\star i_0]_f}$ with argument $i_1 = i - 1$. Notice that the IH is given as $\forall_{(\|t_0\|_f<\|t\|_f,\, t'_0,\, 0<i_1\leq\|t_0\|_f)}\exists j_1, \nabla \vdash t_{0(i_1)_f} \approx_{\{A,AC\}} t'_{0(j_1)_f}$and $\nabla \vdash t_{0[\star i_1]_f} \approx_{\{A,AC\}} t'_{0[\star j_1]_f}$. Then, applying IH, a witness j is obtained such that, with the pre-conditions: $\|f\,t_{[\star 1]_f}\|_f < \|t\|_f$ and $\nabla \vdash f\,t_{[\star 1]_f} \approx_{\{A,AC\}} f\,t'_{[\star i_0]_f}$, one obtains $\nabla \vdash f\,t_{(i)_f} \approx_{\{A,AC\}} f\,t'_{(j)_f}$ and $\nabla \vdash f\,t_{[\star(i)]_f} \approx_{\{A,AC\}} f\,t'_{[\star j]_f}$. The first pre-condition is solved by an application of the definition of $\|\_\|$ and an auxiliary lemma for the operators $\|t\|_f$ and $t_{[\star i]_f}$. The second is exactly the assumption. Then one just needs to consider two cases: $i_0 \leq j_1$ or $i_0 > j_1$. One instantiates j respectively as $j_1 + 1$ or j_1 and concludes using properties of the operators $\|t\|_f, t_{(i)_f}$ and $t_{[\star i]_f}$.

Lemma 4.10 (Transitivity of $\approx_{\{A,AC\}}$) *If $\nabla \vdash t_1 \approx_{\{A,AC\}} t_2$ and $\nabla \vdash t_2 \approx_{\{A,AC\}} t_3$ then $\nabla \vdash t_1 \approx_{\{A,AC\}} t_3$.*

The formalisation is by induction on the size of t_1, where the size of the specified nominal terms is given by their components according to the data structure built

inductively from their syntax. The terms t_2 and t_3 are generalised, and inversions from the equational inference rules are applied to both $\nabla \vdash t_1 \approx_{\{A,AC\}} t_2$ and $\nabla \vdash t_2 \approx_{\{A,AC\}} t_3$. The difficult cases are those of rules [$\approx_\alpha$-ab$_2$] and [`equiv_A`] or [`equiv_AC`]. For [$\approx_\alpha$-ab$_2$], an interesting subcase is when $a \neq a' \neq a'_0 \neq a$: the premisses are $\nabla \vdash t \approx_{\{A,AC\}} (a\,a')\,t' \wedge \nabla \vdash a \# t'$ and $\nabla \vdash t' \approx_{\{A,AC\}} (a'\,a'_0)\,t'_0 \wedge \nabla \vdash a'_0 \# t'_0$, the IH is given as $\forall_{(s_1,s_2,s_3)}, |s_1| < |t| \wedge (\nabla \vdash s_1 \approx_{\{A,AC\}} s_2 \wedge \nabla \vdash s_2 \approx_{\{A,AC\}} s_3) \Rightarrow \nabla \vdash s_1 \approx_{\{A,AC\}} s_3$, and one should conclude that $\nabla \vdash [a]t \approx_{\{A,AC\}} [a'_0]t'_0$. Applying [$\approx_\alpha$-ab$_2$] it remains to prove that $\nabla \vdash a \# t'_0$ and $\nabla \vdash t \approx_{\{A,AC\}} (a\,a'_0)\,t'_0$. The former is obtained by freshness preservation, and the latter by IH with application of Lem. 4.5, equivariance and freshness preservation.

For rules [`equiv_A`] or [`equiv_AC`], the following context is reached at some point of the formalisation, where for the case of [`equiv_A`], $i = i_0 = 1$: the premisses are $\nabla \vdash t_{(1)_{f_k^E}} \approx_{\{A,AC\}} t'_{(i)_{f_k^E}} \wedge \nabla \vdash f_k^E\, t_{[\star 1]_{f_k^E}} \approx_{\{A,AC\}} f_k^E\, t'_{[\star i]_{f_k^E}}$, and $\nabla \vdash t'_{(1)_{f_k^E}} \approx_{\{A,AC\}} t'_{0(i_0)_{f_k^E}} \wedge \nabla \vdash f_k^E\, t'_{[\star 1]_{f_k^E}} \approx_{\{A,AC\}} f_k^E\, t'_{0[\star i_0]_{f_k^E}}$, the IH is given by $\forall_{(s_1,s_2,s_3)}, |s_1| < |f_k^E\, t| \wedge (\nabla \vdash s_1 \approx_{\{A,AC\}} s_2 \wedge \nabla \vdash s_2 \approx_{\{A,AC\}} s_3) \Rightarrow \nabla \vdash s_1 \approx_{\{A,AC\}} s_3$, and one should conclude that $\nabla \vdash f_k^E\, t \approx_{\{A,AC\}} f_k^E\, t'_0$. Applying [`equiv_A`] and the IH the case in which $E = A$ is easily proved. When $E = AC$, one uses Lem. 4.9 and the second premise above, obtaining a third premise: $\exists i_1, \nabla \vdash t'_{(i)_{f_k^E}} \approx_{\{A,AC\}} t'_{0(i_1)_{f_k^E}} \wedge \nabla \vdash t'_{[\star i]_{f_k^E}} \approx_{\{A,AC\}} t'_{0[\star i_1]_{f_k^E}}$. Then, applying [`equiv_AC`] instantiated with i_1, the resulting subgoals are $\nabla \vdash t_{(1)_{f_k^E}} \approx_{\{A,AC\}} t'_{0(i_1)_{f_k^E}}$ and $\nabla \vdash f_k^E\, t_{[\star 1]_{f_k^E}} \approx_{\{A,AC\}} f_k^E\, t'_{0[\star i_1]_{f_k^E}}$, and from the first and third premises above, both subgoals are solved.

Lemma 4.11 (Symmetry of $\approx_{\{A,AC\}}$) *If $\nabla \vdash t \approx_{\{A,AC\}} t'$ then $\nabla \vdash t' \approx_{\{A,AC\}} t$.*

The formalisation is by induction on $\approx_{\{A,AC\}}$ applying lemmas 4.5, 4.8 and 4.10, freshness preservation and equivariance. In particular, the use of Lem. 4.10 is crucial: in the [$\approx_\alpha$-`ab`$_2$] case one should poof that $\nabla \vdash [b]t' \approx_{\{A,AC\}} [a]t$ having as hypotheses $\nabla \vdash t \approx_{\{A,AC\}} (a\,b)\,t'$ and $\nabla \vdash a \# t'$, with IH $\nabla \vdash (a\,b)\,t' \approx_{\{A,AC\}} t$. Then, Lem. 4.10 is applied twice instantiating t_2 as $(a,b)\,t$ and as $(a\,b)\,(a\,b)\,t'$, that allows the use of Lemmas 4.5 (with properties of $\approx_\alpha$) and equivariance to conclude.

To check $\approx_{\alpha,A}$ and $\approx_{\alpha,AC}$ one uses the following corollary. Remember that $\approx_{\alpha,A}$, $\approx_{\alpha,AC}$ and $\approx_{\{A,AC\}}$ are specified as `equiv`$(\{0\})$, `equiv`$(\{1\})$ and `equiv`$(\{0,1\}$.

Corollary 4.12 *For $S \subseteq \{0,1\}$, `equiv`(S) is also an equivalence relation.*

The formalisation is obtained by the manipulation of the superscritps in $S^{-1} = \{0,1\} - S$. For a general equivalence problem `equiv`$(S)(\nabla, t_1, t_2)$, one replaces all superscripts of the operators in the terms t_1 and t_2 inside the set S^{-1} for new ones that neither belong to $\{0,1\}$ nor occur in t_1 and t_2 obtaining respectivelly t'_1 and t'_2. Then, by induction on the inference rules for `equiv`, one easily proves that `equiv`$(S)(\nabla, t_1, t_2) \Leftrightarrow$ `equiv`$(S)(\nabla, t'_1, t'_2) \Leftrightarrow$ `equiv`$(\{0,1\})(\nabla, t'_1, t'_2)$. Thus, using that `equiv`$(\{0,1\})$ is an equivalence relation one concludes.

5 Algorithms for general $\approx_{\alpha,A}$, $\approx_{\alpha,AC}$, $\approx_{\{A,AC\}}$ problems

This section is concerned with checking the validity of α-equivalence constraints in the presence of A and AC symbols, by applying simplification rules.

For example, using the simplification rules given in [26], a constraint of the form $[a]X \approx_\alpha [b]X$ reduces to the set of constraints $a \# X$, $b \# X$; therefore, $a \# X, b \# X \vdash [a]X \approx_\alpha [b]X$. Similarly, assuming $+$ is an AC function symbol, the equality $\nabla \vdash +\langle s, +\langle t, [a]X\rangle\rangle \approx_{\alpha,AC} +\langle +\langle [b]X, s\rangle, t\rangle$ holds whenever the freshness constraints $a \# X$, $b \# X$ belong to ∇. Equational problems are written as pairs (∇, P), where ∇ is a set of freshness constraints and P a set of equations.

The complexity of checking validity of α-equivalence constraints has been studied in [10], where an algorithm to test α-equivalence of nominal terms (both ground or non-ground), derived from a *core algorithm* to solve matching problems modulo α, is provided. The matching algorithm is linear in the size of the problem, when adopting "lazy permutations", for the ground case (i.e., when matching a term s against a ground term t) and therefore α-equivalence is also linear in this case. If both terms are non-ground, then α-equivalence is log-linear in the size of the problem, whereas matching is log-linear if the pattern is linear and quadratic otherwise.

The mutually recursive functions `Check` and $\texttt{Check}_{\texttt{AC}}$ (Algorithms 1 and 2) give the algorithm for checking α-equivalence of a problem (∇, P) modulo A and AC.

Remark 5.1 Lines 10 to 14, regarding the application of the rule $[\approx_\alpha\texttt{-ab}_2]$, have a secondary check for freshness constraints in $a \# t'$. This requires an algorithm for validating freshness constraints based on simplification rules for freshness (Table 1 bottom up) which is linear in $(\nabla, a \# t')$. To avoid repeatedly computations, (for instance the check for $a \# t'$ may appear several times in the computation) one could append valid freshness constraints in ∇, that is, line 12 becomes $\texttt{Check}(\nabla \cup \{a \# t'\}, \{s' \approx_{\{A,AC\}} (a\,b)t'\} \cup P')$. Special care has to be taken with $(a\ b) \cdot t'$ (line 12, rule $[\approx_\alpha\texttt{-ab}_2]$), since it is not a term in our syntax, the permutation has to be propagated in t' and this introduces an aditional linear factor on the complexity of checking α-equivalence. However, adopting the approach in [10], where the syntax is enlarged with suspended permutations over terms and they are propagated in a "lazy" way, this linear factor is avoided obtaining a log-linear algorithm for α-equivalence.

Example 5.2 Assuming $\nabla = \{a \# X, b \# X\}$ it follows that

$$
\begin{aligned}
&(\nabla, \{[a]g\langle a, X\rangle \approx_\alpha [b]g\langle b, X\rangle\}) \Longrightarrow_{Line\,12} (\nabla, \{g\langle a, X\rangle \approx_\alpha (a\,b)g\langle b, X\rangle\})\\
&= (\nabla, \{g\langle a, X\rangle \approx_\alpha g\langle a, (a\,b)X\rangle\}) \Longrightarrow_{Line\,34} (\nabla, \{\langle a, X\rangle \approx_\alpha \langle a, (a\,b)X\rangle\})\\
&\Longrightarrow_{Line\,8} (\nabla, \{a \approx_\alpha a, X \approx_\alpha (a\,b)X\}) \Longrightarrow_{Line\,6,16} (\nabla, \emptyset) \Longrightarrow \top
\end{aligned}
$$

Algorithm 2 deals with equations headed by AC-function symbols. The call $\texttt{Check}_{\texttt{AC}}(\nabla, f_k^{AC} s' \approx_{\{A,AC\}} f_k^{AC} t', 1)$ in line 30 of Algorithm 1 will check equality of the first argument on the left-hand side of the equation with the first, second, third, etc. of the right-hand side until this check succeeds, and also check of the equality of the whole terms eliminating the first argument on the left-hand side and the respective i^{th} argument of success on the right-hand side, otherwise the search

Algorithm 1 Checking for α-equivalence modulo A and AC

function $\texttt{Check}(\nabla, P)$
  if $P = \emptyset$ **then** $\top$
  else let $s \approx_{\{A,AC\}} t \in P$ and $P' = P \setminus \{s \approx_{\{A,AC\}} t\}$ **in**
  case $s \approx_{\{A,AC\}} t$ **of**
   $\langle\rangle \approx_{\{A,AC\}} \langle\rangle$ **:** $\texttt{Check}(\nabla, P')$ ▷ rule [$\approx_\alpha$-ut]
   $a \approx_{\{A,AC\}} a$ **:** $\texttt{Check}(\nabla, P')$ ▷ rule [$\approx_\alpha$-at]
   $\langle s_1, s_2\rangle \approx_{\{A,AC\}} \langle t_1, t_2\rangle$ **:**
    $\texttt{Check}(\nabla, \{s_1 \approx_{\{A,AC\}} t_1, s_2 \approx_{\{A,AC\}} t_2\} \cup P')$ ▷ rule [$\approx_\alpha$-pr]
   $[a]s' \approx_{\{A,AC\}} [a]t'$ **:** $\texttt{Check}(\nabla, \{s' \approx_{\{A,AC\}} t'\} \cup P')$ ▷ rule [$\approx_\alpha$-ab$_1$]
   $[a]s' \approx_{\{A,AC\}} [b]t'$ **:**
    if $\nabla \vdash a \# t'$ **then**
     $\texttt{Check}(\nabla, \{s' \approx_{\{A,AC\}} (a\,b)\cdot t'\} \cup P')$ ▷ Remark 5.1
    else $\bot$
    end if ▷ rule [$\approx_\alpha$-ab$_2$]
   $\pi.X \approx_{\{A,AC\}} \pi'.X$ **:**
    if For all $a \in ds(\pi,\pi'), a \# X \in \nabla$ **then** $\texttt{Check}(\nabla, P')$
    else $\bot$
    end if ▷ rule [$\approx_\alpha$-su]
   $f_k^A s' \approx_{\{A,AC\}} f_k^A t'$ **:**
    let $n_s = ||s'||_{f_k^A}$ and $n_t = ||t'||_{f_k^A}$ **in**
    if $n_s \neq n_t$ **then** $\bot$
    else
     if $\texttt{Check}(\nabla, \{(f_k^A s')_{(1)_{f_k^A}} \approx_{\{A,AC\}} (f_k^A t')_{(1)_{f_k^A}}\})$ **then**
      if $n_s = 1$ **or** $\texttt{Check}(\nabla, \{(f_k^A\ s)_{[\star 1]_{f_k^A}} \approx_{\{A,AC\}} (f_k^A\ t)_{[\star 1]_{f_k^A}}\})$ **then** $\texttt{Check}(\nabla, P')$
      else $\bot$
      end if
     else $\bot$
     end if
    end if
   $f_k^{AC} s' \approx_{\{A,AC\}} f_k^{AC} t'$ **:**
    if $\texttt{Check}_{\texttt{AC}}(\nabla, f_k^{AC} s' \approx_{\{A,AC\}} f_k^{AC} t', 1)$ **then** $\texttt{Check}(\nabla, P')$
    else $\bot$
    end if
   $f_k^E s' \approx_{\{A,AC\}} f_k^E t'$ **:** $\texttt{Check}(\nabla, \{s' \approx_{\{A,AC\}} t'\} \cup P')$ ▷ rule [equiv_Fc]
   _ **:** $\bot$ ▷ otherwise
  end if
end function

continues recursively.

Algorithm 2 Checking α-equivalence modulo A and AC - AC-function symbol case

function $\texttt{Check}_{\texttt{AC}}(\nabla, f_k^{AC} s \approx_{\{A,AC\}} f_k^{AC} t, i)$
  if $||s||_{f_k^{AC}} \neq ||t||_{f_k^{AC}}$ **then** $\bot$ ▷ Check the length of the tuples
  else if not $1 \leq i \leq ||s||_{f_k^{AC}}$ **then** $\bot$
  else ▷ apply rule [equiv_AC]
   if $\texttt{Check}(\nabla, \{(f_k^{AC}\ s)_{(1)_{f_k^{AC}}} \approx_{\{A,AC\}} (f_k^{AC}\ t)_{(i)_{f_k^{AC}}}\})$ **then**
    if $||s||_{f_k^{AC}} = 1$ **or** $\texttt{Check}(\nabla, \{(f_k^{AC}\ s)_{[\star 1]_{f_k^{AC}}}) \approx_{\{A,AC\}} (f_k^{AC}\ t)_{[\star i]_{f_k^{AC}}})\})$ **then** $\top$
    else $\texttt{Check}_{\texttt{AC}}(\nabla, f_k^{AC} s \approx_{\{A,AC\}} f_k^{AC} t, i+1)$
    end if
   else $\texttt{Check}_{\texttt{AC}}(\nabla, f_k^{AC} s \approx_{\{A,AC\}} f_k^{AC} t, i+1)$
   end if
  end if
end function

Example 5.3 $\texttt{Check}(\emptyset, \{f_k^{AC}([a]a, \pi\cdot X) \approx_{\{A,AC\}} f_k^{AC}(\pi'\cdot Y, [b]b)\})$ calls $\texttt{Check}_{\texttt{AC}}$,

which will proceed as follows:

$$\begin{aligned}
&\texttt{Check}_{\texttt{AC}}(\emptyset, f_k^{AC}\langle [a]a, \pi\cdot X\rangle \approx_{\{A,AC\}} f_k^{AC}\langle \pi'\cdot Y, [b]b\rangle, 1)\\
&\Longrightarrow_{Line\,9} \texttt{Check}_{\texttt{AC}}(\emptyset, f_k^{AC}\langle [a]a, \pi\cdot X\rangle \approx_{\{A,AC\}} f_k^{AC}\langle \pi'\cdot Y, [b]b\rangle, 2)\\
&\Longrightarrow_{Line\,6} \texttt{Check}(\emptyset, \{f_k^{AC}\ \pi\cdot X \approx_{\{A,AC\}} f_k^{AC}\ \pi'\cdot Y\}),\ \text{ since } [a]a \approx_{\{A,AC\}} [b]b\\
&\Longrightarrow_{Line\,31,\,Alg.\,1} \texttt{Check}_{\texttt{AC}}(\emptyset, f_k^{AC}\ \pi\cdot X \approx_{\{A,AC\}} f_k^{AC}\ \pi'\cdot Y, 1)\\
&\Longrightarrow_{Line\,5} \texttt{Check}(\emptyset, \{\pi\cdot X \approx_{\{A,AC\}} \pi'\cdot Y\}) \Longrightarrow_{Line\,35,\,Alg.1} \bot
\end{aligned}$$

Theorem 5.4 *Checking the validity of α-equivalence modulo A and AC of a problem of the form (∇, P)*

(i) *can be done log-linearly in (∇, P), whenever the problem does not include AC-function symbols;*

(ii) *and has complexity bounded by the size (i.e., $|(\nabla, P)| = |\nabla| + |P|$, where $|P|$ is the sum of the size of terms in equations in P and $|\nabla|$ is the number of atoms and variables occurring in ∇) to the fourth, otherwise.*

Proof. (sketch) To obtain these complexities we assume the use of suspended permutations over terms and of lazy propagation of permutations (see Remark 5.1). **(i)** For simplicity consider a problem of the form $(\nabla, \{s \approx_{\{A,AC\}} t\})$ where s and t do not contain AC-function symbols. For all maximal subterms of s and t that are headed by A-function symbols one can linearly pre-compute their arguments. This can be done using sequences of arrays of terms in which arguments of A-functions are *flattened*. The lines 30 to 33 of the function `Check`, that correspond to the AC case, will never be executed since the input problem has not and will not generate other problems with AC symbols. The other lines correspond to the α-equivalence verification, except those lines for the case of A-function symbols: lines 19 to 29. For the A case, since arguments were pre-computed the problem can be directly decomposed, similarly to rule `[≈-pr]`, into a new problem with n_s new disjunct equational subproblems, that is a problem of the form $(\nabla, P \cup \{f_k^A s' \approx_{\{A,AC\}} f_k^A t'\})$ becomes directly a problem of the form $\nabla, P \cup \{s'_{(1)_{f_k^A}} \approx_{\{A,AC\}} t'_{(1)_{f_k^A}}, \ldots, s'_{(n_s)_{f_k^A}} \approx_{\{A,AC\}} t'_{(n_s)_{f_k^A}}\})$. The analysis proceeds as for syntactic α-equivalence.
(ii) Let $(\nabla, \{s \approx_{\{A,AC\}} t\})$ be a problem that contains AC-function symbols. Assuming the flat representation of all maximal subterms of s and t that are headed with AC-function symbols is pre-computed, lines 30-33 of `Check`, verify if for subterms s' and t' in s and t headed by an AC-function symbol, say f_k^{AC}, the tuple of arguments in s' contains arguments that are related by α-equivalence modulo AC to arguments of the tuple of arguments in t'. These arguments are not necessarily in the same positions in the tuples of arguments of s' and t'. In the worst case scenario, for each argument of the tuple of arguments regarding f_k^{AC} in s', say $s'_{(i)_{f_k^{AC}}}$, the procedure has to go over the whole tuple regarding f_k^{AC} in t', checking $(\nabla, \{s'_{(i)_{f_k^{AC}}} \approx_{\{A,AC\}} t'_{(j)_{f_k^{AC}}}\})$, for $i, j \leq ||s'||_{f_k^{AC}}$. In case it is true, the algorithm eliminates these two arguments, and repeats the procedure for the remaining arguments of the tuples of arguments of s' and t'. For the steps of `Check` (except by

lines 30 - 33) one already knows that the procedure is log-linear on (∇, P). The complexity of $\texttt{Check}_{\texttt{AC}}$ essentially falls in the problem of searching a perfect matching in the bipartite graph that consists of vertices V labelled by the n_s arguments of the left and the right-hand sides and edges, E, between vertices labelled with terms that match, as it was proved in [6], using the usual first-order syntax. This problem is known to have solutions of complexity $O(|V|\,|E|)$, that is the same as $O(|V|^3)$ since in the worst case one has $O(|V|^2)$ edges [13]. One concludes that searching for a perfect matching is bounded cubically on the size of the problem, since the number of arguments, $||s'||_{f_k^{AC}}$, is linearly bounded in size of the problem. □

Note that the algorithm proposed can check validity of α-equivalence constraints modulo A and/or AC ($\approx_{\{A,AC\}}$) with multiple occurrences of function symbols, some that might be A and some AC, at once. This is due to the fact that different function symbols are not compared and also that distributive properties from one symbol to the other are not considered.

6 Conclusion and Future Work

The soundness of nominal α-equivalence and its extension to the equational theories A, AC and $A + AC$ were formalised in Coq. The grammar of nominal terms was specified in such a way that in addition to A and AC rules one can easily add other inference rules to express properties such as *idempotency* (I), *neutral* (U) and *inverse* elements (**G**roup theory), and their combinations A, AC, AI, ACI, ACU, ACUI, etc.

Enriching nominal α-equality with equational theories formally, will provide an effective framework for dealing not only with nominal α-equivalence, but also with other related fundamental relations such as nominal *unification* and *narrowing* in concrete applications. Examples of such applications can be found in several contexts such as the one of integrity of cryptography protocols [1,14,15]. A further interesting analysis would be the classification of the related "nominal α-unification modulo" problems regarding their unification type and complexities. In particular, a nominal α-unification problem might give rise to infinite solutions.

References

[1] M. Ayala-Rincón, M. Fernández, and D. Nantes-Sobrinho. Nominal narrowing. In *LIPIcs Proc. 1st Int. Conf. on Formal Structures for Computation and Deduction (FSCD)*, vol. 52, pages 11:1 –11:17, 2016.

[2] M. Ayala-Rincón, M. Fernández, and A. C. Rocha-oliveira. Completeness in PVS of a Nominal Unification Algorithm. In *Logical and Semantic Frameworks with Appl. 2015 (LSFA)*, ENTCS 323:57-74, 2016.

[3] B. Aydemir, A. Bohannon, and S. Weirich. Nominal Reasoning Techniques in Coq. *Electronic Notes in Theoretical Computer Science*, 174(5):69–77, 2007.

[4] F. Baader, W. Snyder, P. Narendran, M. Schmidt-Schauß, and K. U. Schulz. *Unification Theory.* Handbook of logic in artificial intelligence and logic programming, 2001.

[5] H. Barendregt. The Lambda Calculus: Its Syntax and Semantics, revised ed., vol. 103 of Studies in Logic and the Foundations of Mathematics, 1984.

[6] D. Benanav, D. Kapur, and P. Narendran. Complexity of Matching Problems. *J. Symb. Comput.*, 3(1/2):203–216, 1987.

[7] A. Boudet, E. Contejean, and H. Devie. A New AC Unification Algorithm with an Algorithm for Solving Systems of Diophantine Equations. *Fifth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 289–299, 1990.

[8] W. E. Byrd and D. P. Friedman. αKanren: A Fresh Name in Nominal Logic Programming. *In Proc. of the Workshop on Scheme and Functional Programming*, pages 79–90, 2007.

[9] C. F. Calvès and M. Fernández. Implementing Nominal Unification. *ENTCS*, 176(1):25–37, 2007.

[10] C. F. Calvès and M. Fernández. Matching and alpha-equivalence check for nominal terms. *Journal of Computer and System Sciences*, 76(5):283 – 301, 2010.

[11] J. Cheney. α Prolog Users Guide & Language Reference Version 0.3 DRAFT, 2003.

[12] E. Copello, Á. Tasistro, N. Szasz, A. Bove, and M. Fernández. Principles of Alpha-Induction and Recursion for the Lambda Calculus in Constructive Type Theory. In *Logical and Semantic Frameworks with Applications 2015 (LSFA)*, ENTCS 323:109-124, 2016.

[13] T. H. Cormen, C. E. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[14] V. Cortier, S. Delaune, and P. Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1):1–43, 2006.

[15] S. Escobar, C. Meadows, and J. Meseguer. Maude-npa: Cryptographic protocol analysis modulo equational properties. *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009*, 5705:1–50, 2007.

[16] M. Fernández and M. J. Gabbay. Nominal rewriting. *Information and Comp.*, 205(6):917–965, 2007.

[17] M. Fernández and M. J. Gabbay. Closed nominal rewriting and efficiently computable nominal algebra equality. *Elect. Proc. in Theoretical Computer Science*, 34:37–51, 2010.

[18] M. Fernández, M. J. Gabbay, and I. Mackie. Nominal rewriting systems. *Int. conference on Principles and practice of declarative programming*, pages 108–119, 2004.

[19] R. Kumar and M. Norrish. (Nominal) Unification by Recursive Descent with Triangular Substitutions. *Interactive Theorem Proving*, pages 51–66, 2010.

[20] A. M. Pitts. Nominal Logic : A First Order Theory of Names and Binding. *Int. Symposium on Theoretical Aspects of Computer Software*, 2215:219–242, 2001.

[21] M. Schmidt-Schauß, T. Kutsia, J. Levy, and M. Villaret. Nominal Unification of Higher Order Expressions with Recursive Let. RISC Report Series TR no. 16-03, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, 2016.

[22] M. R. Shinwell. The Fresh Approach: functional programming with names and binders. Technical report, University of Cambridge, 2005.

[23] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. *Int. Conference on Functional Programming*, pages 263–274, 2003.

[24] C. Urban. Nominal Techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

[25] C. Urban. Nominal Unification Revisited. *Int. Work. on Unification*, pages 513–527, 2010.

[26] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.